Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite

Aashish Phansalkar Ajay Joshi Lizy K. John
ECE Department
The University of Texas at Austin
{aashish, ajoshi, ljohn@ece.utexas.edu}

ABSTRACT

The recently released SPEC CPU2006 benchmark suite is expected to be used by computer designers and computer architecture researchers for pre-silicon early design analysis. Partial use of benchmark suites by researchers, due to simulation time constraints, compiler difficulties, or library or system call issues is likely to happen; but a random subset can lead to misleading results. This paper analyzes the SPEC CPU2006 benchmarks using performance counter based experimentation from several state of the art systems, and uses statistical techniques such as principal component analysis and clustering to draw inferences on the similarity of the benchmarks and the redundancy in the suite and arrive at meaningful subsets.

The SPEC CPU2006 benchmark suite contains several programs from areas such as artificial intelligence and includes none from the electronic design automation (EDA) application area. Hence there is a concern on the application balance in the suite. An analysis from the perspective of fundamental program characteristics shows that the included programs offer characteristics broader than the EDA programs' space. A subset of 6 integer programs and 8 floating point programs can yield most of the information from the entire suite.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes, Modeling techniques, Design studies

General Terms

Measurement, Performance, Design, Experimentation

Keywords

Benchmark subset, clustering, SPEC CPU2006

1. Introduction

SPEC, since its formation in 1988, has served a long way in developing and distributing technically credible real-world application-based benchmarks for computer vendors, computer architects, researchers and consumers. The SPEC CPU benchmark suite, which was first released in 1989 as a collection of ten compute-intensive benchmark programs, is now in its fifth generation and has grown to 29 programs. In order to keep pace with the technological advancements, compiler improvements, and emerging workloads, in each generation of SPEC CPU benchmark suites, new programs are added, programs susceptible to unfair compiler optimizations are retired, program run times are increased, and memory access intensity of programs is increased [4][11][24]. The SPEC CPU2006 benchmark suite comprises of 12 integer and 17 floating point compute-intensive programs for measuring the performance of a processor, memory subsystem, and compiler.

A poorly chosen set of benchmark programs may not accurately depict the true performance of a processor design. On one hand, selecting too few benchmarks may not cover the entire spectrum of applications that may be executed on a computer system; while on the other hand, selecting too many similar programs will increase evaluation time without providing additional information. Therefore, in order to reduce the benchmarking effort, a benchmark suite should have programs that are representative of a wide range of application areas without having many programs with similar characteristics. Understanding similarity between programs can help in selecting benchmark programs that are distinct, but are still representative of the target workload space.

The microprocessor report from October 2006 presents an article [18] which analyzes the SPEC CPU2006 benchmark suite, and raises a question,

"Is SPEC CPU2006 well-balanced?"

There are several programs from certain application areas - for example, in the integer suite, there are 3 programs (`458.sjeng`, `445.gobmk`, `473.astar`) from the Artificial Intelligence area, and in the floating point suite, there are 4 Fluid Dynamics programs (`410.bwaves`, `434.zeusmp`, `437.leslie3d`, `470.lbm`), but no benchmarks from Electronic Design Automation (EDA) application area. The previous generation SPEC CPU suites contained EDA applications, `175.vpr` and `300.twolf` (CPU00), `espresso` and `eqntott` (CPU89 and 92). Is losing these applications that are representative of EDA workloads, a weakness of CPU2006? Or, do some other programs included in the suite have characteristics similar to the EDA programs? When multiple programs from one area are included, is there sufficient uniqueness to warrant their inclusion? In this paper, we analyze these issues based on (dis)similarity between fundamental performance characteristics of benchmarks. The article in microprocessor report [18] also discusses redundancy in benchmark suites. It states –

ISCA'07, June 9-13, 2007, San Diego, California, USA.

"In truth, rather than too few programs, the several SPEC CPU suites have tended to contain too many programs: that is, they invariably comprehend redundant programs that add little or nothing to the mixture of operations represented and whose inclusion or exclusion makes little or no difference to the overall score achieved."

How true is this about SPEC CPU2006? How much redundancy is there in the CPU2006 suite? Analysis of (dis)similarity between programs can also help in addressing this question. Also, partial use of benchmark suite is common for simulation based studies. Citron [2] [3] conducted a survey on benchmark subsets used by computer architecture researchers in top computer architecture conferences and showed that partial use of subsets can lead to incorrect and misleading inferences. The information about similarity between programs can be used to identify a representative subset of programs and their input sets, as opposed to a random selection of a partial suite.

In this paper we apply multivariate statistical analysis techniques such as Principal Components Analysis (PCA) and cluster analysis to (i) study the balance of the CPU2006 benchmark suite, (ii) identify similarity/dissimilarity between CPU2006 programs, (iii) identify similarity between multiple input sets and find representative ones, and (iv) propose subsets of CPU2006 programs that are representative of the whole set. The characterization of programs in this paper is based on run-time program profile information and measurements from five different state-of-the-art machines that represent the most popular architectures.

The remainder of this paper is organized as follows. Section 2 gives an overview of the SPEC CPU2006 benchmark suite and its instruction stream characteristics. Section 3 describes the methodology used to measure redundancy between programs and proposes a representative subset of programs and inputs sets. Section 4 studies the application balance in the SPEC CPU2006 suite and compares them to programs from SPEC CPU2000. Section 5 surveys related research work and in Section 6 we conclude with a summary of the key results from this study.

2. OVERVIEW OF SPEC CPU2006

The SPEC CPU2006 suite, like its predecessors is divided into two parts: the integer component (CINT2006 benchmarks) and the floating point component (CFP2006 benchmarks). The integer group consists of 12 programs, written in C and C++, and the floating point group consists of 17 programs written in C, C++, and FORTRAN languages. In this section, we provide an overview of the runtime characteristics of SPEC CPU2006 integer and floating-point benchmarks in terms of their instruction mix and instruction locality. These runtime characteristics are measured on a Pentium D processor (2.1 GHz, 16KB L1 data and instruction caches, and 2x2MB L2 cache) system running `SUSE Linux 10.1` and were compiled using `Intel C/C++, and FORTRAN compiler V9.1`. The performance counter measurements were carried out using the PAPI [5] tool set.

2.1 Instruction Mix

Table 1, shows the dynamic instruction count and the instruction mix of the programs. The dynamic instruction count of 24 out of the 29 benchmarks is of the order of a few trillion instructions, as compared to a maximum of few hundred billion instructions per program in the CPU2000 suite – further exacerbating the problem of simulation time.

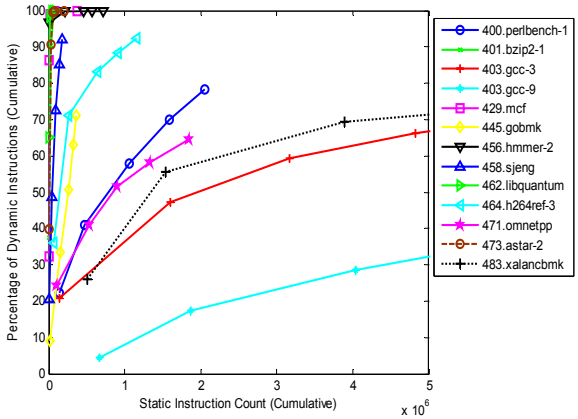
Table 1: Dynamic Instruction Count and Instruction Mix of SPEC CPU2006 Integer and Floating-Point Benchmarks.

Name – Language	Inst. Count (Billion)	Branches	Loads	Stores
CINT 2006				
400.perlbench –C	2,378	20.96%	27.99%	16.45%
401.bzip2 – C	2,472	15.97%	36.93%	12.98%
403.gcc – C	1,064	21.96%	26.52%	16.01%
429.mcf –C	327	21.17%	37.99%	10.55%
445.gobmk –C	1,603	19.51%	29.72%	15.25%
456.hmmer –C	3,363	7.08%	47.36%	17.68%
458.sjeng –C	2,383	21.38%	27.60%	14.61%
462.libquantum-C	3,555	14.80%	33.57%	10.72%
464.h264ref- C	3,731	7.24%	41.76%	13.14%
471.omnetpp- C++	687	20.33%	34.71%	20.18%
473.astar- C++	1,200	15.57%	40.34%	13.75%
483.xalancbmk- C++	1,184	25.84%	33.96%	10.31%
CFP 2006				
410.bwaves – Fortran	1,178	0.68%	56.14%	8.08%
416.gamess – Fortran	5,189	7.45%	45.87%	12.98%
433.milc – C	937	1.51%	40.15%	11.79%
434.zeusmp–C,Fortran	1,566	4.05%	36.22%	11.98%
435.gromacs-C, Fortran	1,958	3.14%	37.35%	17.31%
436.cactusADM-C, Fortran	1,376	0.22%	52.62%	13.49%
437.leslie3d – Fortran	1,213	3.06%	52.30%	9.83%
444.namd – C++	2,483	4.28%	35.43%	8.83%
447.dealII – C++	2,323	15.99%	42.57%	13.41%
450.soplex – C++	703	16.07%	39.05%	7.74%
453.povray – C++	940	13.23%	35.44%	16.11%
454.calculix –C, Fortran	3,041	4.11%	40.14%	9.95%
459.GemsFDTD – Fortran	1,420	2.40%	54.16%	9.67%
465.tonto – Fortran	2,932	4.79%	44.76%	12.84%
470.lbm – C	1,500	0.79%	38.16%	11.53%
481.wrf - C, Fortran	1,684	5.19%	49.70%	9.42%
482.sphinx3 – C	2,472	9.95%	35.07%	5.58%

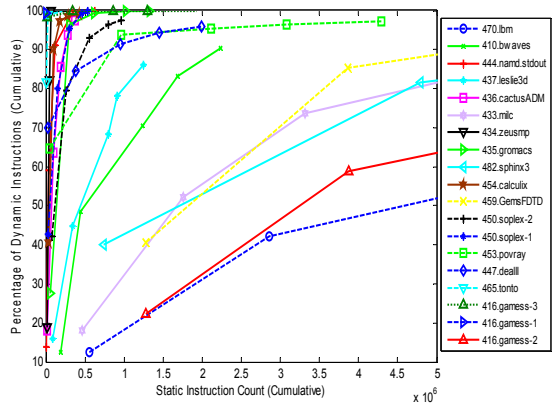
There are several interesting observations from the instruction mix of the programs. For integer programs, the percentage of branches in the dynamic instruction stream is close to the typical 20%. However, two programs, `456.hmmer` and `464.h264ref` only have 7% branches. In `483.xalancbmk`, one of three C++ programs in the integer suite has 25% branches. In comparison, the other two C++ programs, `471.omnetapp` and `473.astar`, are more typical with 20% and 15% branch instructions respectively. Among the floating-point programs, `447.dealII`, `450.soplex` and `453.povray` have approximately 15% branches where as most of the other floating-point programs have less than 5% branch instructions. There are a few floating-point programs, `410.bwaves`, `470.lbm`, `436.cactusADM`, which have less than 1% branches. The large average dynamic basic block size in these programs suggests a high degree of parallelism that can be exploited by out-of-order microarchitectures.

2.2 Instruction Locality Based on Subroutine Profiling

In order to understand the code locality in the CPU2006 programs, we perform subroutine profiling using the PIN dynamic instrumentation tool [17]. PIN can identify hot subroutines based on subroutine call frequency. It can also count the number of dynamic/static instructions in the subroutines. Figure 1 shows the locality characteristics for at least one input set of integer and floating point benchmarks. Appendix I shows a table with the summary of data measured for this experiment. The cumulative percentage of dynamic instructions executed by a program is shown on Y-axis and the cumulative count of static instructions is shown on the X-axis with a log scale. The first point in the line plot for each benchmark represents the hottest subroutine - X-coordinate shows the number of static instructions in the routine and Y-axis the percentage of dynamic instructions that it represents. The second, third, fourth and fifth points respectively represent the top five, ten, fifteen and twenty hot subroutines. Many programs initially show a steep upward climb as the static instruction count increases, which suggests very good instruction locality.



(a) CINT2006 Benchmarks



(b) CFP2006 Benchmarks

Figure 1. Instruction locality based on code reuse in the top 20 hot subroutines for SPEC CPU2006 benchmarks.

As shown in Figure 1 the top twenty subroutines cover 80% or more of the dynamic instructions in almost all benchmarks. The integer benchmark `456.hmmer` shows a very high reuse of code in the hottest subroutine. More than *95%* of the instructions come from the hottest subroutine which has *11,080* static instructions. Similarly the floating point benchmarks `436.CactusADM` and `470.lbm` show a very high code–reuse and hence good instruction locality. On the other hand, the 20 hot subroutines in `471.omnetpp`, `483.xalancbmk` and `403.gcc` account for a very low percentage of dynamic instructions, suggesting a relatively poor instruction locality. The trend observed in the previous generation SPEC CPU benchmark suites [19] continues with `gcc` exhibiting the poorest instruction locality among all the benchmark programs. In SPEC CPU2006, `403.gcc-9` has the poorest instruction locality with 5 million static instructions only accounting for approximately 45% of the dynamic instructions. As such, we can conclude that similar to the programs in the previous generation SPEC CPU benchmark suites, most of the SPEC CPU2006 benchmarks exhibit good instruction locality, but there are some notable exceptions with very large instruction footprints and relatively poor instruction locality.

Apart from the instruction locality, there are other microarchitectural attributes which can be measured during execution of the benchmark programs. Table 2 shows the range of some important characteristics measured using performance monitoring counters on a Pentium D system described earlier in this section, illustrating the diversity of the benchmarks in the CPU2006 suite. Many characteristics are seen to vary orders of magnitude between the minimum and maximum.

Table 2: Range of important performance characteristics of SPEC CPU2006 benchmarks

Metric	Min	Max
I-cache miss ratio	~ 0	1.7%
L1 D-cache miss ratio	6.3%	33%
L2 cache misses per instruction (per L2 access)	~0 (0.01%)	2.4% (49%)
DTLB miss ratio	0.2%	8.4%

3. REDUNDANCY IN SPEC CPU2006 SUITE

Although the SPEC CPU2006 benchmark suite comprises of programs from different application domains, it is possible that they exhibit similar program characteristics. In this section we outline the methodology to measure the (dis)similarity in fundamental program characteristics of benchmarks. We then apply this technique to measure the redundancy of programs in the SPEC CPU2006 benchmark suite, and propose a subset of representative programs and input sets that can be used if the time required to simulate the entire benchmark suite is prohibitive.

3.1 Methodology

We measured fundamental program characteristics related to their instruction locality, data locality, branch predictability, and instruction mix, using hardware performance counters. These characteristics are microarchitecture-dependent and the results could be biased by the idiosyncrasies of a particular machine. Therefore, in order to eliminate this bias we measured the program characteristics on five different state-of-the-art machines with four different Instruction Set Architectures (ISAs) and compilers (IBM Power, Sun UltraSPARC, Itanium, and x86).

Table 3 shows the list of performance counter based characteristics that were measured for each program on five different machines. We performed a correlation analysis between every performance counter characteristic and Cycles-Per-Instruction (CPI), and only selected characteristics that showed a good correlation to performance. This process eliminates the performance counters that exhibit a large variation but have little or no impact on performance. Note that the important metrics that affect performance for the integer and floating-point programs are different. In addition to these characteristics, the variability in microarchitectures, ISAs, and compilers across the five machines helps in capturing the differences between the benchmarks. The hardware performance counter data used in this study was measured by various members of the SPEC CPU subcommittee members on their state-of-the-art machines. Also, the methodology outlined in this section was used by the SPEC CPU subcommittee as one of the factors in evaluating the candidates for the SPEC CPU2006 benchmark suite [12]. The confidentiality requirements prevent us from disclosing the details of the machines on which this data was measured nor the performance counter data from individual machines.

Table 3. Program Characteristics used for measuring similarity between Integer and Floating-Point programs.

Integer benchmarks	Floating-Point benchmarks
Integer operations per instruction	Floating point operations per instruction
L1 instruction cache misses per instruction	Memory references per instruction
Number of branches per instruction	L2 data cache misses per instruction
Number of mispredicted branches per instruction	L2 data cache misses per L2 accesses
L2 data cache misses per instruction	Data TLB misses per instruction
Instruction TLB misses per instruction	L1 data cache misses per instruction

Since the measurements are carried out on five different machines, each performance counter characteristic-machine pair is treated as a variable. If we have *n* machines and we measure *m* characteristics for each machine, we have *n x m* variables for each program. There is a pitfall of directly using these raw variables to measure similarity between programs. It is possible that some of these variables are correlated and measure the same inherent benchmark property. Therefore, using a large number of correlated variables will unduly overemphasize the importance of a particular benchmark property. In order to remove the correlation between these variables, this dataset is pre-processed using Principal Components Analysis (PCA) [7]. Clustering Analysis (CA) is then used to group programs with similar program characteristics. We now describe the two statistical analysis techniques – Principal Component Analysis and Cluster Analysis.

3.1.1 Removing Correlation using PCA

Considering the six characteristics measured on each of the five different machines, we have thirty characteristics per program. It is humanly impossible to simultaneously look at all the data and draw meaningful conclusions from them. Hence PCA is used to analyze the data. In order to isolate the effect of varying ranges of each parameter, the data is first normalized to a unit normal distribution, *i.e.* a normal distribution with mean equal to zero and standard deviation equal to 1, for each variable. PCA helps to reduce the dimensionality of a data set while retaining most of the original information. PCA computes new variables, so-called principal components, which are linear combinations of the original variables, such that all the principal components are uncorrelated. PCA transforms *p* variables X_1, X_2,...., X_p into *p* principal components (PC) $Z_1,Z_2,\ldots,Z_p$ such that:

$$Z_i = \sum_{j=0}^{p} a_{ij} X_j$$

This transformation has the property *Var* $[Z_1] \geq$ Var $[Z_2] \geq \ldots \geq$ Var $[Z_p]$ which means that Z_1 contains the most information and Z_p the least. Given this property of decreasing variance of the PCs, we can remove the components with the lower values of variance from the analysis. This reduces the dimensionality of the data set while controlling the amount of information that is lost. We use a standard technique (Kaiser Criterion) to choose PCs where only the top few PCs which have eigenvalues greater than or equal to one are retained. In order to capture the entire information from p original variables, p PCs may be required, but if there are several correlated variables, a small number of PCs can capture most of the information from the original variables.

3.1.2 Measuring Similarity using Cluster Analysis

Clustering is a statistical technique that can be used to group programs with similar features. There are two commonly used clustering techniques – K-means clustering and hierarchical clustering. The K-means clustering algorithm divides a set of N programs into K groups, where K is a value specified by the user. Therefore, in order to evaluate different grouping possibilities one needs to cluster programs for different values of K and then select the best fit. On the other hand, as the name suggests, hierarchical clustering is useful in simultaneously looking at multiple clustering possibilities and the user can select the desired number of clusters using a dendrogram. Therefore, in this paper we use hierarchical clustering algorithm. Hierarchical clustering is a bottom up approach and starts with a matrix of distance between N cases or benchmarks. The distance is the Euclidean distance between the program characteristics. The algorithm used for hierarchical clustering is as follows:

1. Assign each program to its own cluster, such that if we have N programs we have N clusters.
2. Find the closest (most similar) pair of clusters and merge them into a single cluster, so that we have one less cluster.
3. Compute distances (similarity) between the new cluster and the old cluster.
4. Repeat steps 2 and 3 until all items are grouped into a single cluster of size N.

This hierarchical clustering process can be represented as a tree or dendrogram, where each step in the clustering process is illustrated by a joint in the tree (e.g.: Figure 2). The numbered scale on the horizontal axis corresponds to the linkage distance between programs. We now apply this methodology to find a representative subset of programs from the SPEC CPU2006 benchmark suite.

3.2 Subsetting SPEC CPU2006 Benchmarks

To keep pace with advancements in technology and the increase in size of on-chip caches, the data footprint and run time of SPEC CPU benchmark programs has been significantly increased. However for architectural studies that use cycle-accurate simulators, it is almost impossible to simulate all programs and input sets in a reasonable amount of time. If the same amount of information can be obtained from a smaller subset of representative programs, it would certainly help architects and researchers to cut down the simulation time without compromising on the inferences drawn from their studies.

This section demonstrates the result of applying PCA and cluster analysis for selecting a subset of benchmark programs when an architect or researcher is constrained by time and wants to select a reduced subset of programs from the suite. Figure 2 shows a dendrogram for CINT2006 benchmarks obtained after applying PCA and Hierarchical Clustering on the performance counter data from Table 3. The Euclidean distance between the benchmarks is used as a measure of dissimilarity and single-linkage distance is computed to create a dendrogram. Seven Principal Components (PCs) with eigen values greater than one are chosen and they retain 94% of the variance. In the dendrogram in Figure 2 the horizontal axis shows the linkage distance indicating the dissimilarity between the benchmarks. The ordering on the Y-axis does not have particular significance, except that benchmarks are positioned close to each other when the distance is smaller. Benchmarks that are outliers have larger linkage distances with the rest of the clusters formed in a hierarchical way. One can use this dendrogram to select a representative subset of programs. For example, if a researcher wants to reduce his simulation budget to just six benchmarks, then drawing a vertical line at linkage distance of 4, as shown in Figure 2, will give a subset of six benchmarks *(k=6)*. Drawing a line at a point close to 4.5 yields a subset of four benchmarks *(k=4)*. Table 4 shows the resulting subsets of the CINT2006 suite. In clusters where there are more than two programs, the representative of cluster *i.e.* the benchmark closest to the center of the cluster is chosen as a representative. As we traverse from left to right on the dendrogram the number of benchmarks in the subset keep decreasing. This helps the user to select appropriate benchmarks when simulation time is a constraint.

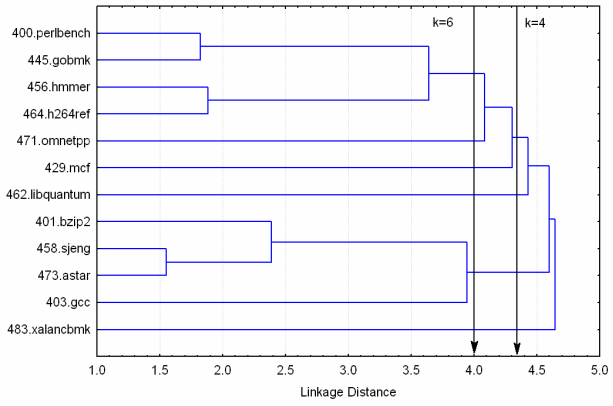


Figure 2. Dendrogram showing similarity between CINT2006 Programs.

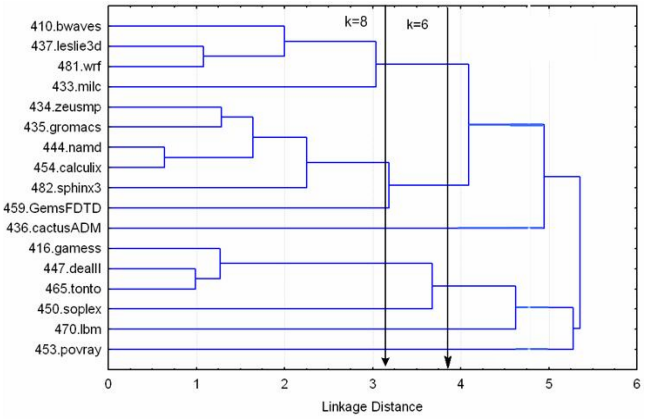


Figure 3. Dendrogram showing similarity between CFP2006 Programs.

Figure 3 shows the dendrogram for floating point benchmarks in CPU2006. Five PCs are chosen using the Kaiser criterion which retains 85% of the variance. The two vertical arrows show the points at which the subsets of size 6 and 8 are formed. The resulting clusters are shown in Table 5. The distance of each of the benchmarks in the cluster to the cluster center has to be recalculated and a representative can be chosen. In Figure 3 there are two main clusters which split at extreme right because the branch characteristics of the benchmarks, `447.dealII`, `450. soplex and`, `453.povray` exhibit a comparatively higher branch misprediction rate. In the next section we evaluate the representativeness of these subsets.

One should note that clustering and subsetting gives importance to unique features and differences. It helps to eliminate redundancy and duplicated efforts in experimentation. However, one should not mistake the mix of program types in a subset as the mix of program types in real-world workloads.

Table 4. Representative subset of SPEC CINT2006 programs.

Subset of Four Programs	`400.perlbench, 462.libquantum,473.astar,483.xalancbmk`
Subset of Six Programs	`400.perlbench, 471.omnetpp, 429.mcf, 462.libquantum, 473.astar, 483.xalancbmk`

Table 5. Representative subset of SPEC CFP2006 programs.

Subset of Six Programs	`437.leslie3d, 454.calculix, 436.cactusADM, 447.dealII, 470.lbm, 453.povray`
Subset of Eight Programs	`437.leslie3d, 454.calculix, 459.GemsFDTD,436.cactusADM, 447.dealII, 450.soplex, 470.lbm, 453.povray`

3.3 Evaluating Representativeness of Subsets

We evaluate the usefulness of the subsets, proposed in the previous section, to estimate the speedup of the entire suite on

eight different commercial systems. The SPEC web page[1] presents performance results of SPEC CPU2006 benchmarks on commercial computer systems. We used these results to evaluate the efficacy of the subset of programs proposed in the previous section. We obtained the execution times for all benchmarks on 8 different platforms and their execution times on a reference machine. We then compared the weighted average (geometric mean) speedup from the subset against the average (geometric mean) speedup from the entire component (CINT or CFP) of the suite. When calculating the average speedup from the subset, each benchmark of the subset was assigned a weight proportional to the number of benchmarks in its cluster.

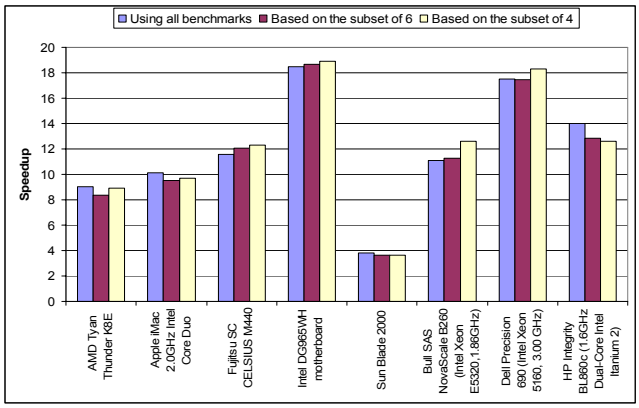


Figure 4. Validation of CINT2006 subset using performance scores of eight systems from the SPEC CPU website.

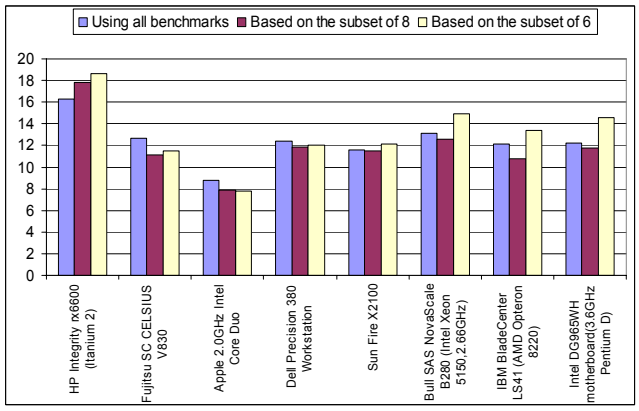


Figure 5. Validation of CFP2006 subset using performance scores of eight systems from the SPEC CPU website.

Figure 4 shows the comparison for CINT2006 benchmarks using the subset of 4 and 6 benchmarks shown in Table 4. For CINT component the subset of 4 programs shows an average error of 5.8% and a maximum error of 10.1%. The subset of 6 benchmarks shows an average error of 3.8% and a maximum error of 8%. This shows that even a subset of 4 benchmarks out of 12 CINT benchmarks has a very good predictive power in estimating the speedup shown by the entire suite.

[1] *http://www.spec.org/cpu2006/results/cpu2006.html*

Figure 5 shows the validation of CFP2006 benchmarks using the subsets from Table 5. The maximum error in the floating point subset of 6 is higher than that in the integer benchmark subset. For a subset of 6 the average error is 10.8% with the maximum error of 19%. Hence we look at a subset of 8 benchmarks which shows the average error of 7% and the maximum error is 12%. From these results, we observe that 6 out of 12 integer benchmarks and 8 out of 17 floating point benchmarks form a good representative subset. It is interesting to observe that a third or half of the benchmarks in a suite can contain most of the information in the entire suite.

3.4 Selecting representative input sets

Many benchmarks in the CPU2006 have multiple input sets. For example, `403.gcc` benchmark has nine input sets. A reportable SPEC result for each benchmark is supposed to comprise of all its input sets. However, for simulation based studies, researchers typically select one input set. Instead of selecting an input set in an ad hoc manner, clustering analysis can also be used to select a representative input set. The program characteristics shown in Table 3 were measured for all the different benchmarks and input sets. PCA and clustering analysis was performed on this data to find similarity between input sets of each benchmark.

Figure 6 shows the dendrogram for input sets and the benchmarks for the integer component. The Kaiser criterion results in choosing seven PCs covering 89% of variance for this analysis. Some benchmarks have only one input set and are hence represented only by their name. In some benchmarks, all input sets appear clustered together, where as in many cases, some input sets are very different from the other input sets of the same benchmark. As an example, the behavior of `403.gcc-9` is significantly different from its sibling input sets.

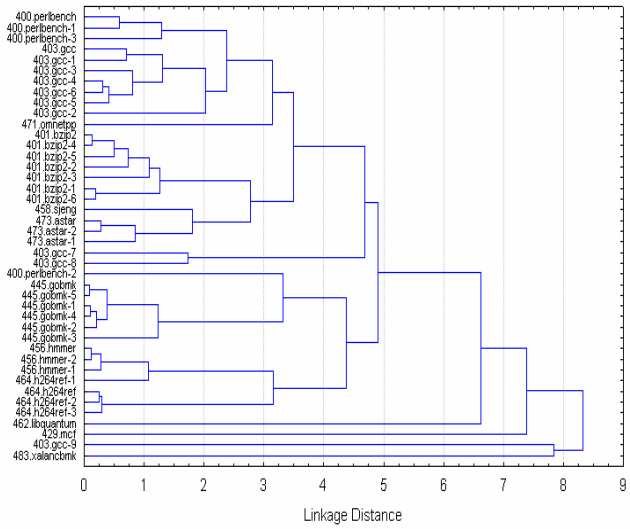


Figure 6. Dendrogram showing similarity between program-input set for each benchmark in the SPEC CPU2006 suite.

In this analysis, a benchmark's input set closest to the whole (aggregated) benchmark run is selected as the representative input set. In CINT2006, the benchmarks that have multiple input sets

are 400.perlbench, 401.bzip2, 403.gcc, 445.gobmk, 456.hmmer, 464.h264ref and 473.astar. Similarly, we also perform the analysis for input sets of CFP2006 programs. Figure 7 shows the dendrogram showing similarity between inputs sets of programs from CFP2006. Six PCs covering 88% of variance are chosen. In this category there are only two benchmarks with multiple input sets - 416.gamess and 450.soplex. For each of these benchmarks we list the most representative input set in Table 6. This information will be useful in selecting the most representative input set for each program.

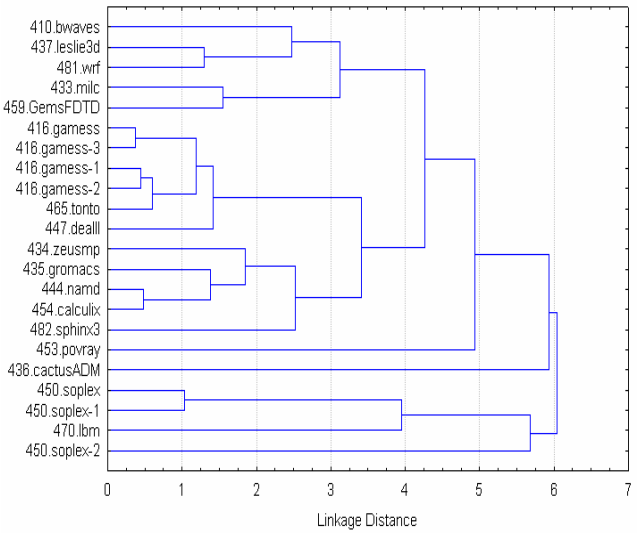


Figure 7. Dendrogram showing similarity between program-input set for each program from CFP2006.

Table 6. List of representative input sets for SPEC CPU2006 programs.

CINT2006 benchmarks	464.h264avc - input set 2
400.perlbench - input set 1	473.astar - input set 2
401.bzip2 - input set 4	
403.gcc - input set 1	**CFP2006 benchmarks**
445.gobmk - input set 5	416.gamess - input set 3
456.hmmer - input set 2	450.soplex - input set 1

3.5 Subsets based on branch and memory access characteristics

Often, researchers focus on optimizing the design to take advantage of certain program characteristics. In this section, we separately analyze the similarity of programs based on data access performance characteristics and branch prediction characteristics. This similarity information can be used when conducting data cache and branch prediction studies. In this section we analyze all programs in the suite without classifying them into CINT and CFP groups. Figures 8 shows the scatter plot based on the first 2 PCs of the branch characteristics, covering approximately 92% of the variance.

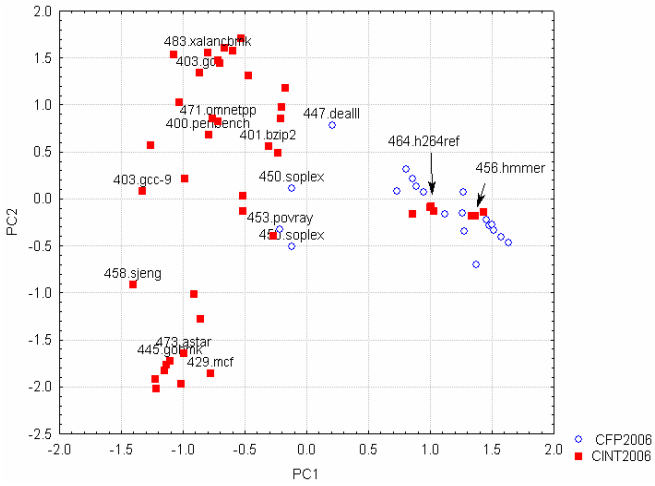


Figure 8. CINT and CFP programs in the PC workload space using branch predictor characteristics.

The CFP benchmarks are clustered together but the CINT programs clearly show diversity. A few CINT workloads overlap with the cluster of CFP workloads on the right side of the plot. These CINT benchmarks (464.h264ref, 456.hmmer) have fewer branches like the other floating programs around them. Majority of the floating point programs have very little diversity in their branch characteristics. The benchmarks with high value of PC1, e.g. majority of CFP benchmarks and 464.h264ref and 456.hmmer show easy to predict branches. On the other hand CINT benchmarks are more spread out in the space and show significantly diverse behavior. The data used for this analysis was measured from three different systems.

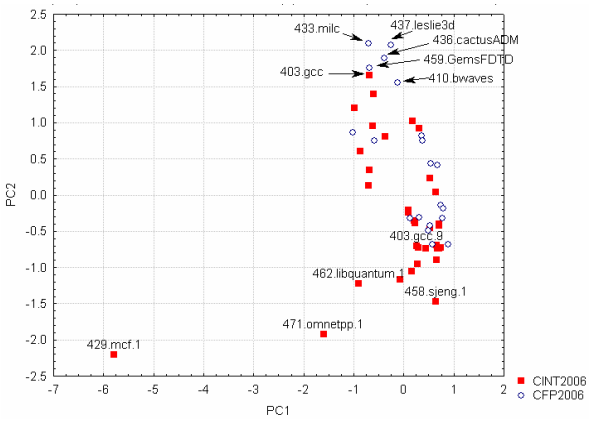
The data accesses characteristics were also measured on three different systems and after applying PCA, 4 PCs were retained which account for 84% of the variance. Figure 9 shows a scatter plot based on the first 4 PCs. For the purpose of clarity only certain benchmarks are labeled in the scatter plots. The benchmarks that have a more negative value of PC1 *e.g.* 429.mcf, 471.omnettp, 462.libquantum exhibit a poor data access behavior and hence result in higher data cache miss-rates. The CFP benchmarks that are located at the top show very high percentage of memory accesses and hence should also be considered when selecting benchmarks for cache studies. In summary, when selecting benchmarks for a certain study the user should look at the workload space of those characteristics and select benchmarks to ensure that the entire workload is covered. Only selecting outliers will exercise worst case or best case behavior and can lead to misleading conclusions.

4. BALANCE IN THE SPEC CPU2006 SUITE

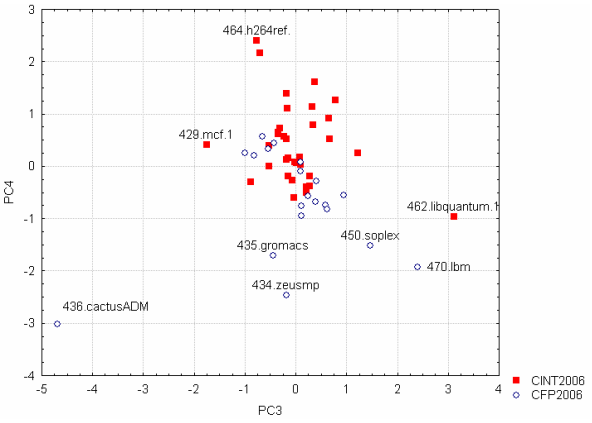
4.1 Case study on EDA Applications

The concern in the October 2006 microprocessor report article [18] on CPU2006, whether the suite is balanced, stems from the fact that certain application areas have multiple representative programs (see Table 7), whereas certain areas are not represented in the suite. For example, in the integer suite, there are 3 programs (458.sjeng, 445.gobmk, 473.astar) from the artificial intelligence area, and in the

floating point suite there are 4 programs (410.bwaves,434.zeusmp, leslie3d, lbm) from the fluid dynamics area. In this section we provide a case study on the characteristics of the Electronic Design Automation (EDA) applications. The CPU2006 suite contains none, where as the earlier SPEC CPU suites contained more than one EDA applications (vpr, twolf, espresso, eqntott). Is losing the applications from the EDA area a weakness of CPU2006? Are other programs from other application areas similar to EDA applications that their absence is not an issue? We perform a similarity analysis to understand this. Program characteristics are measured for all the SPEC CPU2000 integer programs and projected in the workload space after performing PCA on the characteristics of CINT2006 benchmarks. Figure 10 shows the projections of the workload space. From these figures it is evident that the EDA tool benchmarks in CPU2000 lie close to 473.astar and 401.bzip2 from CPU2006 benchmarks. Since the EDA programs are well within the envelope of the workload space covered by the new suite, the elimination of EDA programs may not be a major concern.

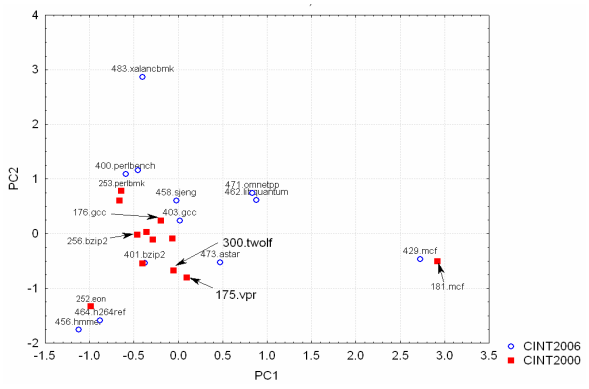


(a) PC1 Vs. PC2

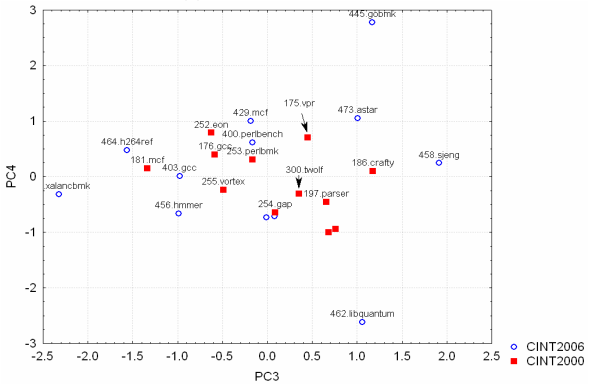


(b) PC3 Vs. PC4

Figure 9. CINT and CFP programs plotted in the PC space using memory access characteristics (PC1 Vs. PC2).



(a) PC1 Vs. PC2



(b) PC3 Vs. PC4.

Figure 10. Scatterplot showing position of EDA applications in the workload space.

4.2 Differences between benchmarks from the same application area

Any two benchmarks that belong to the same application area can show different behavior on certain architecture. Are the programs from SPEC CPU2006 suite, which belong to the same application area really different? The similarity analysis described in Section 3 can answer this question. Let us consider one application area from Table 7 at a time and refer to Figures 2 and 3 to answer the question. In case of artificial intelligence area, 458.sjeng and 473.astar show very similar behavior and can be found quite close to each other in the workload space, while 445.gobmk is much further away from its siblings. The equation solver applications do not lie close to each other and hence justify their presence in the suite. 410.bwaves and 437.leslied, are relatively close to each other than the other two programs in their application area. Both the programs in molecular dynamics are different and relatively close to each other with the linkage distance of less than 2 between them. 465.tonto and 416.gamess also have a linkage distance of less than 2. On the contrary, the 4 programs from the Engineering and Operational Research domain are significantly different from each other. In Table 7 two programs that belong to the same application area and show similar program characteristics are highlighted in bold.

Table 7. Classification of programs based on application areas.

Application area	Benchmarks
Artificial Intelligence	**458.sjeng,** 445.gobmk**,473.astar**
Equation solver	436.cactusADM, 459.GemsFDTD
Fluid Dynamics	**410.bwaves,**434.zeusmp, **437.leslie3D,** 470.lbm
Molecular Dynamics	**435.gromacs, 444.namd**
Quantum Chemistry	**465.tonto, 416.gamess**
Engineering and Operational Research	454.calculix,447.dealII, 450.soplex, 453.povray

4.3 Comparing CPU2006 benchmarks with CPU2000

SPEC CPU subcommittee took efforts to include challenging applications to stay relevant in the light of fast and powerful emerging machines. Looking back after the selection, how did these changes affect the overall characteristics of programs in CPU2000? Are the programs fundamentally different from the ones in SPEC CPU2000?

From Figure 10 it is evident that the CINT2006 benchmarks are spread farther in the workload space as compared to CINT2000 benchmarks and cover a wider area in the workload space. In the PC1 Vs PC2 scatter plot (Figure 9(a)), many of the CPU 2000 programs are located close to the (0,0), whereas, the new programs such as 483.xalancbmk and 456.hmmer provide coverage for very far away spots in the workload space. In the PC3-PC4 scatter plot (Figure 10(a)), one can easily notice the diversity added by the programs 445.gobmk, 483.xalancbmk, 462.libquantum and 458.sjeng. They extend the envelope of the benchmark space significantly. It is also interesting to note that the benchmarks from CPU2000 suite that were retained e.g. *mcf, bzip2, perl, gcc* show similar behavior as their predecessors. This might mean that only the dynamic instruction count and data footprint of these benchmarks changed but the control flow almost remained the same. The increased data footprint will exercise the big caches in recent processors. Due to space constraints we could not show the scatter plots for the CFP2006 benchmarks but we see that the floating benchmarks in CPU2006 are even more diverse compared to the ones in CPU2000. Overall, one can observe increased diversity in the new suite.

4.4 Sensitivity of programs to performance characteristics

In this section, we present a classification of programs based on their sensitivity to branch predictors and data cache across five machines that were used to do the analysis in Section 3. In order to measure the sensitivity of a program to branch predictor and L1 D-cache configuration, for every machine we ranked programs based on these characteristics. The difference in ranks of a program across all machines is then computed. The resulting number is indicative of sensitivity of that program for a given characteristic.

Table 8 shows classification of benchmarks based on their sensitivity to branch and L1 data-cache configuration. It is organized as follows. For each characteristic (branch misprediction rate and L2 data cache miss-rate) the workloads (program-input pairs) are categorized into one of the low, medium and high ranges. The most striking observation from this is that 462.libquantum, 456.hmmer, and 464.h264ref show the most variation in D-cache misses across the five machines. 456.hmmer shows a lot of variation for branches where as 458.sjeng, 473.astar and 445.gobmk show higher misprediction rates. As observed by Vandierendonck et.al. [23] in CPU2000, we also observed that 409.gcc from CPU2006 ranks relatively low in variation across the five machines used in the experiment. The explanation is that every machine is equally good or equally bad for gcc, eliminating the sensitivity to platforms.

Table 8. Sensitivity of Programs to Branch Misprediction Rate and L1 D-cache Miss-rate across five different platforms.

Branch Prediction	
High	456.hmmer-1, 456.hmmer, 456.hmmer-2
Medium	471.omnetpp, 429.mcf, 473.astar-1, 473.astar, 464.h264ref-1, 473.astar-2, 400.perlbench-1, 401.bzip2-4, 462.libquantum,, 401.bzip2-3, 401.bzip2-2, 400.perlbench, 401.bzip2, 445.gobmk-3, 401.bzip2-1, 464.h264ref, 401.bzip2-5,, 403.gcc-8, 458.sjeng,401.bzip2-6, 403.gcc-4
Low	464.h264ref-3, 445.gobmk, 445.gobmk-1, 445.gobmk-4, 445.gobmk-2, 445.gobmk-5, 400.perlbench-2, 464.h264ref-2, 403.gcc-7, 403.gcc-6, 400.perlbench-3, 483.xalancbmk, 403.gcc-2, 403.gcc-5, 403.gcc-1, 403.gcc, 403.gcc-9, 403.gcc-3
L1 D-cache	
High	462.libquantum, 464.h264ref-2, 464.h264ref-3, 464.h264ref, 456.hmmer-1
Medium	456.hmmer, 456.hmmer-2, 400.perlbench-2, 400.perlbench-3, 445.gobmk-3, 403.gcc-7
Low	400.perlbench, 403.gcc-8, 483.xalancbmk, 473.astar-2, 403.gcc, 400.perlbench-1, 473.astar, 464.h264ref-1, 445.gobmk, 473.astar-1, 445.gobmk-4, 471.omnetpp, 429.mcf, 403.gcc-9, 403.gcc-3, 445.gobmk-2, 401.bzip2-3, 401.bzip2-5, 445.gobmk-1, 403.gcc-6, 403.gcc-5, 401.bzip2-2, 401.bzip2-6, 403.gcc-2, 403.gcc-1, 401.bzip2-1, 401.bzip2, 403.gcc-4, 401.bzip2-4, 445.gobmk-5, 458.sjeng

5. RELATED WORK

In [8] and [9] Eeckhout *et.al* measured a mixture of microarchitecture dependent and microarchitecture independent metrics and presented similarity information of programs. Vandierendonck and Bosschere [23] analyzed the SPEC CPU2000 benchmark suite peak results on 340 different machines representing eight architectures, and used PCA to identify the redundancy in the benchmark suite. They found that a small number of CPU 2000 programs have nearly the same predictive power as the entire suite in ranking machines. Giladi and Ahituv [10] also had a similar approach towards finding a subset of programs. They found that the ten programs of SPEC89 suite could be reduced to six without affecting the SPEC rating. Phansalkar et.al. [20] and Joshi et.al. [15] characterized benchmarks using microarchitecture independent metrics to find a representative subset and study the difference between the previous generations of SPEC CPU benchmarks. The metrics are independent of microarchitecture but compiler and ISA dependent. However, since CPU2006 benchmarks have very long runtime (23 out of 29 benchmarks have more than one trillion instructions), significantly higher time would be required to measure these characteristics. The approach used in this paper

tries to achieve the component of microarchitecture independence by characterizing benchmarks on a wide range of systems with different ISAs, compilers and microarchitectures. Yi et.al [27] compare all the different subsetting approaches and show that use of statistical techniques for subsetting can lead to improved accuracy. Citron [2][3] presented subsets based on use by the computer architecture research community and showed that partial use of suites can lead to misleading results.

6. CONCLUSIONS

There are concerns about program redundancy programs in the SPEC CPU2006 benchmark suite and that benchmarks from some certain commonly used application areas are missing. Using performance counter data from five different state of the art machines, and statistical analysis techniques such as PCA and clustering, we analyze the similarity and balance of the recently released SPEC CPU2006 suite. Dendrograms illustrating the similarity between benchmarks and scatter plots showing the workload space are presented for overall selection of metrics, as well separately for branch and data access metrics.

We show that 6 out of the 12 integer programs and 8 of the 17 FP programs can capture most of the information from the FP benchmarks. It is observed that not all programs in the same application area are similar. Some of them are more similar to the programs in another application area. When analyzed from the perspective of program characteristics, an unrepresented area such as electronic design automation (EDA) may not be a major weakness of the suite. We also identify one input set as a representative input set for programs that have multiple input sets. Not all input sets of a benchmark result in similar behavior. Some input sets make a program appear more similar to another benchmark rather than its own sibling.

It is less than a year since SPEC CPU2006 has been released. If Citron's observation regarding use of benchmarks [2][3] holds, it will be a quite some time before the research community can succeed in compiling and simulating the entire benchmarks. Even if the intentional misuse may not happen, partial use of the suite will be inevitable due to difficulties with compilation, system call issues, libraries etc. The information presented in this paper should help researchers in selecting a representative set of benchmarks, if subsetting is unavoidable. It should also help in understanding and interpreting simulation results.

7. DISCLAIMER

All the observations and analysis done in this paper on SPEC CPU2006 benchmarks are the authors' opinions and should not be used as official or unofficial guidelines from SPEC in selecting benchmarks for any purpose. This paper only provides guidelines for researchers and academic users to choose a subset of benchmarks should the need be.

8. ACKNOWLEDGEMENTS

The authors express their gratitude to the following SPEC member companies (IBM, Sun Microsystems, Intel, AMD, Apple, HP, SGI) for providing performance counter data from their systems. We enjoyed the opportunity to interact with the SPEC CPU Subcommittee and provide clustering analysis during selection of CPU2006 programs [12]. The researchers are supported in part by National Science Foundation under grant number 0429806. Ajay Joshi is supported by an IBM fellowship.

9. REFERENCES

[1] M. Alt "Performance Modeling Using Compilers" White paper Intel Corp. http://cache-www.intel.com/cd/00/00/22/64/226491_226491.pdf

[2] D. Citron, "MisSPECulation: partial and misleading use of spec CPU2000 in computer architecture conferences", Proceedings of the 30th Annual International Symposium on Computer Architecture, pp. 52-59, June 9-11, 2003.

[3] D. Citron, J. Hennessy, D. Patterson, G. Sohi, "The Use and Abuse of SPEC: An ISCA Panel," *IEEE Micro*, vol. 23, no. 4, pp. 73-77, Jul/Aug, 2003.

[4] K. Dixit: Overview of the SPEC Benchmarks. "The Benchmark Handbook", Chapter 9, 1993

[5] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, "Using PAPI for hardware performance monitoring on Linux Systems" Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute, June 2001

[6] J. Dujmovic and I. Dujmovic, "Evolution and Evaluation of SPEC benchmarks", ACM SIGMETRICS Performance Evaluation Review, vol. 26, no. 3, pp. 2-9, 1998.

[7] G. Dunteman, *Principal Components Analysis*, Sage Publications, 1989

[8] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Designing computer architecture research workloads", *IEEE Computer*, 36(2), pp. 65-71, Feb 2003.

[9] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Quantifying the impact of input data sets on program behavior and its applications", Journal of Instruction Level Parallelism, vol 5, pp. 1-33, 2003.

[10] R. Giladi and N. Ahituv, "SPEC as a Performance Evaluation Measure", IEEE Computer, Vol. 28, No. 8, Aug 1995, Pages 33-42

[11] J. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millenium", IEEE Computer, July 2000.

[12] J. Henning. Performance Counters and Development of SPEC CPU2006. *Computer Architecture News.* March 2007

[13] L. John, P. Vasudevan and J. Sabarinathan, "Workload Characterization: Motivation, Goals and Methodology", In Workload Characterization: Methodology and Case Studies, Edited by L. John and A. M. G. Maynard, IEEE Computer Society, pp. 3-14, November 1998

[14] L. John, V. Reddy, P. Hulina, and L. Coraor, "Program Balance and its impact on High Performance RISC Architecture", *Proc. of the International Symposium on High Perf Comp Arch*, pp.370-379, Jan 1995.

[15] A. Joshi, A. Phansalkar, L. Eeckhout, L.K. John "Measuring Benchmark Characteristics Using Inherent Program Characteristics", IEEE Transactions on Computers, Jun2006, Vol 55 No. 6 pp 769-782

[16] T. Lafage and A. Seznec, "Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A

Preliminary Application to the Data Stream", Workshop on Workload Characterization (WWC-2000), Sept 2000.

[17] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, "PIN:Building Customized Program Analysis Tools with Dynamic Instrumentation", Proceedings of 2005 ACM SIPLAN Conference on Programming Language Design and Implementation, pp 190-200, 2005

[18] H. McGhan, SPEC CPU2006 Benchmark Suite, Microprocessor Report, October 10, 2006.

[19] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John, "Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites". IEEE International Symposium on Performance Analysis of Systems and Software. March 2005, pp 10-20

[20] A. Phansalkar, Joshi, A. L. Eeckhout, and L.K. John "Four Generations of SPEC CPU Benchmarks: What has changed and what has not?", *Technical Report TR-041026-1, Laboratory of Computer Architecture, The University of Texas at Austin*. 2004.

[21] J. Reilly. Presentation at IEEE International Symposium on Workload Characterization, Oct 2006 http://www.iiswc.org/iiswc2006/IISWC2006S2.1.pdf

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior", Proc. of International Conference on Architecture Support for Programming Languages and Operating Systems, pp. 45-57, 2002.

[23] H. Vandierendonck, K. Bosschere, "Many Benchmarks Stress the Same Bottlenecks", *Proc. of the Workshop on Computer Architecture Evaluation using Commerical Workloads (CAECW-7)*, pp. 57-71, 2004.

[24] R. Weicker, "An Overview of Common Benchmarks", IEEE Computer, pp. 65-75, Dec 1990.

[25] T. Wenisch, R. Wunderlich, B. Falsafi, and J. Hoe, "Applying SMARTS to SPEC CPU2000", CALCM Technical Report 2003-1, Carnegie Mellon University, June 2003.

[26] J. Yi and D. Lilja, "Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies, and Recommendations," IEEE Transactions on Computers, Vol. 55, No. 3, Mar. 2006, pp. 268-280

[27] J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. Lilja, and L. K. John, "Evaluating Benchmark Subsetting Approaches" International Symposium on Workload Characterization, October 2006, pp 93-104

[28] J.Yi, D. Lilja, and D.Hawkins, "A Statistically Rigorous Approach for Improving Simulation Methodology", Proc. of Intl Conf on High Performance Computer Architecture, Feb 2003, pp 281-291.

APPENDIX I – Subroutine Profile Summary of CPU2006

The columns show the information for the hottest subroutine, the cumulative data for the top 5,10 and 20 hot subroutine. Each of these columns has two sub-columns which show the cumulative percentage of dynamic instructions executed in the respective number of hot routines and the cumulative static count e.g. 22.3% of the dynamic instructions executed are from the hottest subroutine which has 140998 static instructions. Also, the top five hot subroutines account for 41.06% of dynamic instructions which have a total of 466157 static instructions. This data is generated using PIN [17].

Cumulative	Hottest Subroutine		5 Hot Subroutines		10 Hot Subroutines		20 Hot Subroutines	
Benchmark name- input	Percentage Dynamic	Static count	Percentage Dynamic	Static count	Percentage Dynamic	Static count	Percentage Dynamic	Static count
CPU2006 integer benchmarks								
400.perlbench-1	22.30%	140998	41.06%	466157	57.80%	1061283	78.17%	2048489
400.perlbench-2	50.86%	140998	77.58%	722121	82.93%	1396866	88.30%	2673489
400.perlbench-3	67.40%	140998	81.63%	710437	88.43%	1362876	93.70%	2488840
401.bzip2-1	37.42%	3126	90.49%	30930	99.93%	61906	99.99%	107435
401.bzip2-2	34.44%	9357	92.90%	28335	99.98%	61906	100.00%	107435
401.bzip2-3	55.02%	1850	97.34%	29654	99.99%	61906	100.00%	107435
401.bzip2-4	28.53%	9357	91.04%	30930	99.95%	72391	100.00%	207009
401.bzip2-5	51.18%	2022	94.55%	30930	99.98%	72391	100.00%	109398
401.bzip2-6	32.70%	3126	86.96%	30930	99.94%	61906	99.99%	107435
403.gcc-1	16.53%	72483	37.98%	2208290	50.06%	4156689	61.62%	8414654
403.gcc-2	11.61%	72483	25.12%	2323225	35.92%	4131522	49.64%	8041641
403.gcc-3	20.70%	146129	47.27%	1597844	59.24%	3167999	71.23%	6350050
403.gcc-4	20.48%	146129	47.13%	2020198	57.38%	4147096	67.73%	8054140
403.gcc-5	19.21%	146129	49.83%	2124041	59.36%	3929883	69.89%	7651110
403.gcc-6	19.00%	146129	49.01%	2124041	59.79%	3929883	69.93%	7476905
403.gcc-7	16.46%	72483	55.79%	1800707	70.44%	4732572	80.34%	8696609

403.gcc-8	22.37%	72483	54.89%	1473886	64.54%	4272881	74.14%	7894874
403.gcc-9	4.40%	669312	17.42%	1871256	28.45%	4042557	43.22%	7913678
429.mcf	32.31%	2573	86.26%	8137	98.89%	16550	99.97%	375239
445.gobmk-1	8.62%	33206	31.96%	123655	48.10%	262154	67.84%	596812
445.gobmk-2	7.61%	13119	31.45%	123655	47.37%	244954	67.62%	548199
445.gobmk-3	19.04%	32657	40.65%	166002	56.67%	270024	71.23%	494076
445.gobmk-4	7.75%	13119	30.50%	123655	46.86%	224322	66.98%	508799
445.gobmk-5	9.08%	13119	33.48%	150245	50.70%	263013	71.11%	366012
456.hmmer-1	99.10%	11080	99.76%	143630	99.91%	385550	99.96%	993103
456.hmmer-2	96.79%	11080	99.76%	220363	99.99%	458965	100.00%	711948
458.sjeng	20.64%	9213	48.76%	48588	72.63%	97943	92.13%	183112
462.libquantum	65.18%	901	98.38%	12897	100.00%	39182	100.00%	89909
464.h264ref-1	41.21%	63541	75.28%	360800	90.46%	733452	96.06%	1281878
464.h264ref-2	35.20%	63541	65.66%	263448	81.81%	632195	91.08%	1103854
464.h264ref-3	36.20%	63541	71.16%	263448	83.30%	638070	92.28%	1162685
471.omnetpp	24.32%	108195	40.91%	527643	51.53%	903746	64.49%	1849742
473.astar-1	37.86%	5674	81.20%	37331	96.89%	62175	99.47%	283443
473.astar-2	39.82%	5674	90.61%	30981	99.51%	60403	99.92%	205548
483.xalancbmk	25.98%	503940	55.57%	1539078	69.42%	3896705	79.24%	9194199
CPU2006 floating point benchmarks								
410.bwaves	81.72%	955	98.81%	20906	100.00%	206818	100.00%	787062
416.gamess-1	22.30%	1276458	58.80%	3871636	72.88%	7159808	90.66%	14870258
416.gamess-2	40.51%	1276458	85.24%	3865963	94.33%	6848202	98.75%	16678502
416.gamess-3	39.95%	736310	81.60%	4791284	92.23%	8652681	96.80%	15778333
433.milc	17.97%	18724	63.62%	94474	85.66%	190396	97.53%	373321
434.zeusmp	42.67%	24198	80.11%	146870	95.44%	305321	99.75%	537772
435.gromacs	69.95%	28484	84.43%	380679	91.35%	951762	95.95%	1991123
436.cactusADM	98.37%	12299	99.97%	341943	99.99%	865076	100.00%	1931210
437.leslie3d	18.98%	12137	82.03%	43027	99.99%	61553	100.00%	538388
444.namd.stdout	13.76%	7980	59.01%	38959	91.26%	119030	99.99%	623266
447.dealII	18.02%	458978	52.22%	1760935	73.59%	3316274	87.13%	5874499
450.soplex-1	16.02%	88472	44.91%	337648	68.42%	796146	86.16%	1246274
450.soplex-2	42.03%	79680	79.58%	256105	93.06%	559113	97.45%	961334
453.povray	12.63%	185839	48.62%	446779	70.44%	1238650	90.32%	2239198
454.calculix	64.78%	42790	93.73%	964957	95.47%	2111871	97.14%	4292472
459.GemsFDTD	27.49%	58016	96.34%	295624	99.31%	617617	99.92%	1329282
465.tonto	12.48%	554306	42.13%	2856539	59.42%	6679198	76.04%	12522232
470.lbm	99.32%	919	100.00%	8940	100.00%	187934	100.00%	696825
482.sphinx3	40.50%	34299	90.11%	98106	97.13%	174224	99.06%	418795